National Aeronautics and Space Administration (NASA)

OpenSource ESB

Version 4.0

Table of Contents

1. INTRODUCTION

NASA Competency Center (CC) supports all enterprise services, technical services, and system to system integrations on top of the OpenSource ESB platform. This platform is a collection of open source tools and projects that have been pulled together (and sometimes augmented) to provide a complete development, runtime, monitoring, and management toolset for NASA CC to support a SOA. At the center of this platform is the OpenSource ESB which provides the runtime for all services and is built around the JBI implementation ServiceMix [1].

This white paper discusses the technical design and use of the OpenSource ESB.

2. ENTERPRISE SERVICE BUS OVERVIEW

This section details NASA CC's definition of an ESB, its purpose and role, and lays the groundwork for how the OpenSource ESB was designed.

2.1 Enterprise Service Bus Defined

The term Enterprise Service Bus (ESB) is used to describe an architecture that utilizes Web services, messaging middleware, intelligent routing and transformation. An ESB is a lightweight, ubiquitous integration backbone through which software services and application components flow. Since the birth of the ESB concept, there have been a number of suggestions as to what an ESB should be and do. This section will detail how NASA CC defines an ESB and begin to lay the ground work for some of the guiding principles of the OpenSource ESB architecture.

An ESB is a *service enabler*. Its purpose is to expose provide value on top of existing services, data sources, and systems. An ESB should never be used as the originator of data or a service. This means that an ESB should not be used as a general-purpose application server (although it is possible to run an ESB from within an application server). The typical role of an application server (J2EE, SAP, etc.) is to provide a platform upon which source systems can be created. These source systems or service providers "own" the data they operate upon and the services they offer related to this data; an ESB should never own data or services.

An ESB should be implemented with a *decentralized architecture* that allows it be scaled effectively and efficiently to meet all demands placed upon it. This decentralized model is realized in multiple interconnected runtime engines that can be distributed across numerous networks and machines. These interconnected runtimes allow applications to interact with the ESB as if it were a single entity, which means a service can be deployed on any ESB station and then accessed anywhere as part of the "Bus". A side effect of this requirement for a decentralized architecture is that an ESB must have a flexible and loose licensing model that does not hinder runtime distribution to more nodes on the network.

An ESB uses *XML as the standard communication language*. Communication from most clients and some service providers will use XML over various bindings and protocols (SOAP, HTTP, JMS, etc.). Even when XML is not used externally between the ESB and clients or service providers, an ESB processes this non-XML data internally in XML canonical form.

An ESB is built upon and *supports open standards*. An ESB should have strong support for open standard protocols between clients and service providers, such as web services, SOAP, WSDL, HTTP, XML, JMS, and REST. This requirement does not mean that an ESB does not provide proprietary connectors; generally an ESB will have to communicate with legacy systems exposing their services using open standards. The extent to which an ESB provides proprietary connectors is not central to the definition of an ESB. In addition to providing open standards based communication, the ESB framework should both be built upon and provide extensions for using open standards such as JBI, OSGI, and/or JMX. This architectural design point means that services can be moved between ESB implementations with less effort and no dependence on vendor-supplied code, and that developers already familiar with these open standards can be productive much more quickly.

An ESB should have services that can be *easily found and consumable*. A service registry is typically utilized to address this requirement and contains the necessary information required to discover and consume the service. A service registry can be implemented for machine discovery via a UDDI (universal description, discovery, and integration) registry that allows SOAP based interrogation and access to the WSDL documents that describes the protocol bindings and message formats. A service registry can also support a user interface to allow ESB consumers to find and consume the services as needed.

In summary, and ESB is defined as a framework that adheres to the following axioms:

- An ESB is a service enabler, not service provider
- An ESB is implemented using a decentralized architecture
- An ESB uses XML as the standard language for communication
- An ESB supports open standards
- An ESB has services that are easily found and consumable

2.2 ESB Value Added Services

In addition to meeting all the above requirements; an ESB also provides many value added features for the services it exposes. These features are usually orthogonal to the business particulars of a service being exposed and should be implemented in a generic fashion that (usually) does not require customization per each new service or integration. While not exhaustive, the list below provides some of the most common ESB value added features (all of these are supported on the OpenSource ESB platform):

Transformation

An ESB will often be used to transform the content of client requests to a format acceptable to one or more service providers. This pattern of transformation is used so often that it really can not be considered an optional feature for an ESB implementation to implement. The principal benefit of this process is that the ESB can handle multiple, possibly client-specific, versions of a single service, while the service provider operates only on the canonical model. These transformations are normally implemented using XSLT (when dealing with XML) or possible scripting languages such as Ruby.

Protocol Bridging

Protocol bridging is another fundamental value added service that most ESB implementations support. This is the process of transforming a request from one protocol to another. An example of this would be a client making a SOAP request to the ESB, which then transforms the SOAP request to a binary SAP RFC request. When the ESB receives a response from SAP, it then bridges the SAP RFC response back into a XML SOAP response to send back to the client. Many times protocol bridging is used in conjunction with transformation.

Security

A key benefit offered by an ESB is centralized security. An ESB should be nominally equipped with utilities to implement standard security specifications such as WS-Security, SSL, and PKI. Many service providers will lack the tools to implement a custom security solution to the extent possible via an ESB. For example, more and more source systems have the ability to expose web services, but few offer extensive security features. By channeling all services calls through an ESB, security can be implemented across the enterprise landscape in a unified fashion.

Auditing

Auditing is used to record service transactions that occur throughout the ESB. It is normally implemented with logging, events, and archiving. Logging is typically used for debugging purposes and provides a trail for all executed code. Events are summaries of entire transactions that can be used for monitoring purposes. Archiving is the process of recording all requests and responses that occurred for a particular service. These messages can then be reviewed at a later time to verify the transactional content.

Routing

Routing (or content-based routing) is the process of determining the endpoint destination of a service call based upon the content of the service request.

Quality of Service

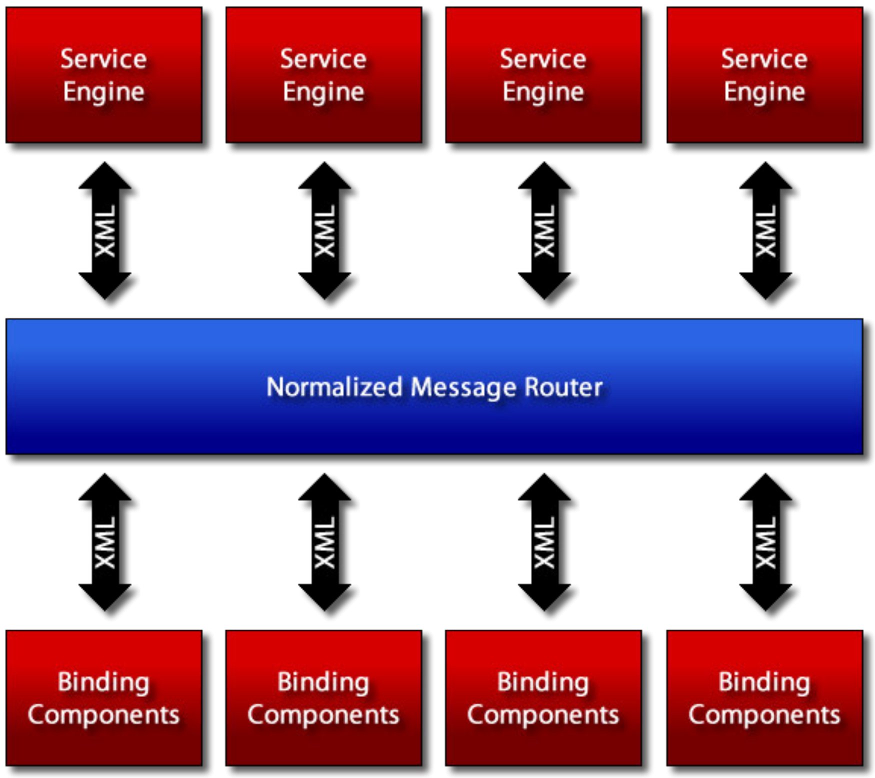
An ESB usually supports a number of message patterns offering quality of service: guaranteed delivery (message persistence), in-order delivery, and once-and-only-once delivery. Guaranteed delivery is used for asynchronous requests where clients can send an event to the ESB for processing by some source system. The ESB will first persist the message, and then acknowledge that the message has been received, and finally process the message. Since the message has been persisted, the ESB can guarantee that the source system will ultimately receive the message, even if the service is currently down. To satisfy in-order delivery, if a sequence of requests is sent to the ESB in some implied order, the ESB will guarantee that this order will be preserved when sending to the source system for processing. In cases where once-and-only-once delivery is promised, an ESB will only send one message to a source system for processing, even when receiving duplicate requests from one or more clients.

Service Level Agreement

Service Level Agreements (SLA) can be implemented on the ESB to establish thresholds for particular services, particular clients, and/or combinations of particular services and clients. Thresholds may include the size of a request (or response), or the time or date that certain requests are made. If certain thresholds are exceeded, the ESB can return an error response to the client without involving the source system.

2.3 Java Business Integration Overview

The Java Business Integration (JBI) is a specification (JSR 208 [2]) that defines a standard, open architecture upon which an ESB can be built and extended. This architecture dictates that third-party components may be "plugged in" to any standard ESB infrastructure then interoperate in a reliable, predictable fashion across multiple vendors' conformant ESB implementations. In the same way that J2EE standardized Java-based application servers, JBI seeks to standardize ESB frameworks.



JBI Component Plug-in Framework

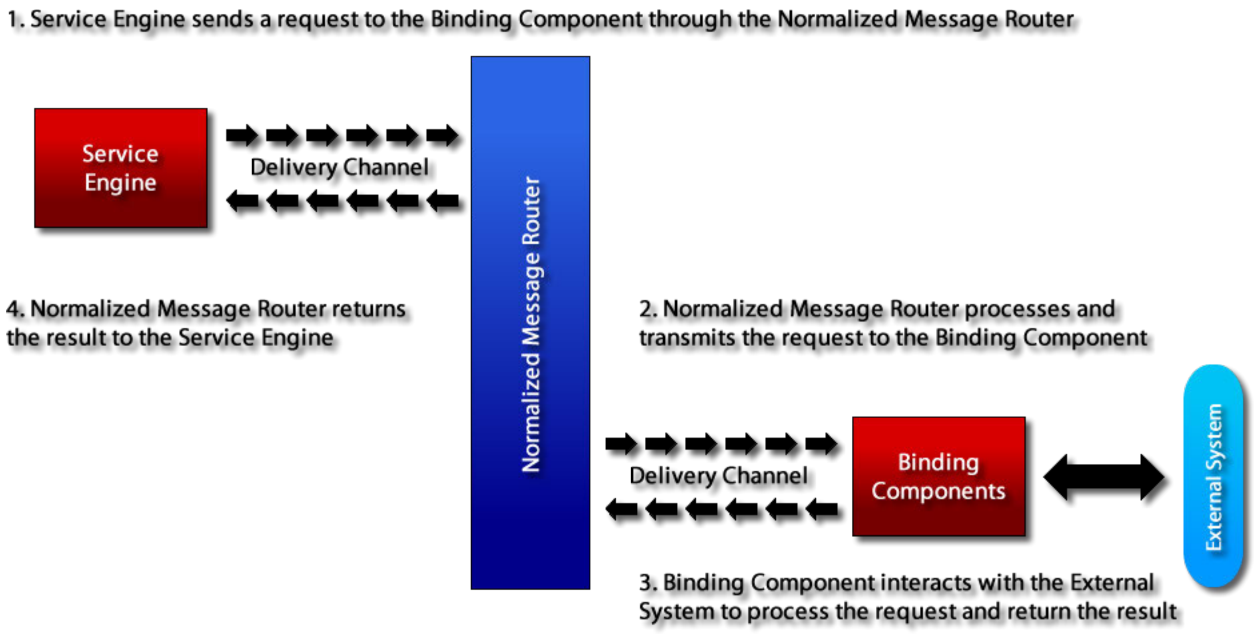
As shown in the diagram above, JBI defines a standard core container; other components, which may in turn be containers themselves, can be plugged into this central container. In effect, JBI can be represented as a container of containers. The JBI specifications dictate how these components interoperate inside the larger container: plug-in components inside a JBI container communicate in a service-oriented fashion via message exchanges based closely on WSDL 2.0.

The plug-in components are divided into two distinct categories: service engines (SE) and binding components (BC).

Service engines provide business logic and transformation services to other components, and may in turn consume other such services. Binding components provide connectivity to services external to a JBI environment. From JBI component API standpoint, there is no distinction between SEs and BCs, however the specification is clear that a logical distinction between the two types of components should be made. Separation of business and processing logical from communications logic reduces implementation complexity, and increases flexibility.

Service engines and binding components can function as service providers, consumers, or both. SEs can integrate Java-based applications (and other resources) or applications with available Java APIs. BCs can integrate applications (and other resources) that use remote-access technology that is not directly available in Java. This connectivity can involve communications protocols or services provided by Enterprise Information Systems (EIS) resources.

The interface for all component interactions is WSDL 2.0. Using this interface promotes a loosely-coupled architecture in which components can be easily exchanged and replaced without affecting the entire system. As shown in the diagram below, components never directly interact with one another; instead, all communication occurs through a normalized message router (NMR). The NMR is the backplane of a JBI runtime environment.


1. Service Engine sends a request to the Binding Component through the Normalized Message Router
Service Engine
Delivery Channel
Normalized Message Router
4. Normalized Message Router returns the result to the Service Engine
2. Normalized Message Router processes and transmits the request to the Binding Component
Delivery Channel
Binding Components
External System
3. Binding Component interacts with the External System to process the request and return the result

JBI Component Interaction

The above diagram is an example of a component exchange through the message router. In this example the SE requires the services of a BC. To initiate communication the SE sends an XML message to the NMR targeted to the BC's endpoint (service address). When the BC receives the request, it performs some communication with an external system (e.g. SAP, Database, etc.) to fulfill its request and sends the response back to the NMR. The NMR then sends the response back to the SE which continues performing its function based on this response. It's important to note that the SE in this example has no particular knowledge of the BC's implementation on external system communication. The SE's only knowledge is through the BC's service definition (WSDL).

The affect of JBI is targeted more for the developers of ESB components then clients of a JBI based ESB. JBI allows for developers to compose new applications and services that can be easily extended later as requirements as change. This benefit is the result of a system integration style imposed by JBI that is similar to a service-oriented approach. Applications will be created from the composition of services as opposed to monolithic code.

The end goal of JBI is to bring an integration middleware platform that prevents vendor lock in (an open architecture). Changing application server or integration middleware vendors will no longer mean abandoning a component's integration technology. In addition, JBI's open, standards-based approach to system integration will likely lead to a wider choice of integration technologies, as you won't be restricted to a single vendor's integration product catalogue.

2.4 Service Registry

2.4.1 Overview

Years after the rise in popularity of Web Services and SOA, it's largely considered that the service registry (UDDI) function of these technologies has been a complete failure [2]. The biggest driver for this is because service registries were originally designed to be machine discoverable and consumable at runtime, which dictated an overly complex and unworkable implementation of a service registry. To address this NEACC has implemented service registry designed for humans, not machines. In addition to meeting the base requirement of ensuring ESB services are easily found and consumable the NEACC service registry has additional entities that construct a complete picture of the NEACC enterprise offering.

2.4.2 NEACC service registry

The NEACC service registry is referred to as MetaHouse. MetaHouse provides the means to keep track of IT related entities throughout NEACC, NASA, and external integrated landscapes. MetaHouse provides a single housing of information of NEACC application and system services that are:

- Open – provide multiple ways to access (HTML, RSS, JSON, XML)
- Searchable – provide ability to search for services
- Subscribe(able) – provide ability to subscribe to service events

MetaHouse is web-based application and is available to NASA internal users at (link removed).

3. OpenSource ESB Platform Architecture

The OpenSource ESB platform incorporates all of the features of an ESB defined in Section 2 in addition to the following features not attributed directly to an ESB runtime:

- Development Tools
- Release Control Tools
- Runtime Monitoring
- Runtime Management

This section will cover the technical details and architecture of this platform.

3.1 Guiding Principles

This first iteration of the OpenSource ESB began as a simple Apache Axis based SOAP implementation in the fall of 2004. Since then there have been numerous changes and open source projects added to the platform. Throughout all these changes (and for changes going forward) the following guiding principles are used to drive all architecture decisions.

Java Based Implementation

The OpenSource ESB is implemented in Java. Java was selected for several important reasons. First, it has one of the largest open source communities associated with any mainstream programming language. By simply electing to use Java, many enterprise-class open source projects are readily available. Second, Java already has a large base of programmers fluent in both the core language and standard libraries. Third by running on top of the Java the ESB can be hosted on various operating systems. Currently the OpenSource ESB runtime has been verified to run on Solaris, Linux, Windows and Mac. Lastly the support for other languages (than Java) that run from within the JVM are increasing in number. This will allow developers to choose whatever programming language (e.g., Ruby, Python, Java, Groovy, Scala) is most applicable for a particular service implementation within the ESB.

Open Source

The use of open source software and tools is an important part of NASA CC platform. Using open source provides several major advantages over a closed or proprietary system.

- No Black Boxes – By using open source software developers have access to the source code and the right to modify it. This right removes constraints on quickly addressing bugs or feature gaps in the underlying platform.
- The Community – Popular open source projects (like the ones the OpenSource ESB is built on) provide excellent support in the form of documentation, user forums and mailing lists. This helps in learning the platform but more importantly developers can get immediate assistance from hundreds of other users and developers when they run into an issue they are not able to solve themselves.
- Emphasis On Open Standards – Due to the freeness of open source software, developers of these platforms are not motivated by profit to promote vender lock-in. This leads to more importance being placed on supporting open, well accepted standards that make it easier to move to different platforms in the future if desired.
- Security – Open source software typically has many more eyes verifying the quality of the software being used then closed source software. It has been proven that security by obscurity (alone) does not work.
- No Distribution Costs – One of the characteristics of an ESB is that it can scale quickly as needed. By using open source software there are no distribution costs associated with the platform to prevent this (or slow it down) from occurring.
- Commercial Support Options – Using open source software does not preclude the option of purchasing commercial support for the platform if desired. There are many companies that provide support and indemnification for common open source platforms.

JBI Compliance

Adherence to the Java Business Integration is a strict requirement for the design of the OpenSource ESB. This specification (Section 2.3) is backed by Sun Microsystems and is considered important for the following important benefits:

- Component Isolation – Interactions between components follow a very strictly defined process through the normalized message router. Isolation between components is enforced down to the thread level (threads are not shared between components). This is important for several reasons. First this implementation isolation promotes reuse. Since implementation details are hidden between components there is less tendency to couple logic across components. Second, this isolation provides for a more fault-tolerant system. A component going down or behaving unexpectedly will have no direct affect on other components that don't require its services. Lastly the independence of components allows for highly concurrent "modules" of code which improves speedup [4] on multi-processor machines.
- Scalability – The message exchange pattern for JBI is modeled after the Staged Event Driven Architecture (SEDA [5]). SEDA increases scalability by reducing the total number of threads required to process increasing requests (while limiting reduction in response time).
- Hot Deployment/Update – JBI provides a very well defined procedures and interfaces for deploying and un-deploying components on the fly without requiring a restart of the ESB. As an ESB becomes the focal point for all system interaction throughout a landscape this feature becomes critical so services (components) can be added, removed, and updated without affecting other services.
- Transaction Visibility – Internal communication between components is accomplished using (binary) XML across the message router. This XML can be easily captured and recorded (auditing) to provide fine grain visibility of what data and processes and particular transaction involves.
- Standards Based – All NEACC developed components adhere strictly to the JBI API and have no dependencies on server specific implementations (ex. ServiceMix). Replacing the JBI implementation with another (Mule, OpenESB, etc.) can be accomplished without changing any component logic.

Lightweight

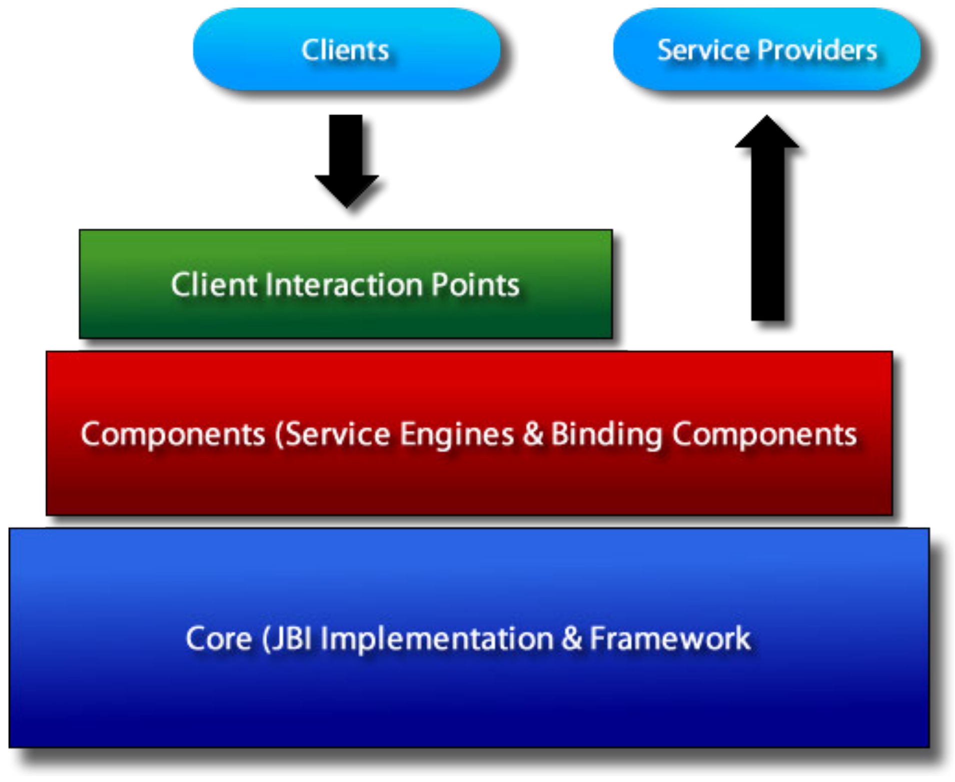
The OpenSource ESB has lightweight architecture: it can be deployed and run in a Java Standard Edition (J2SE) environment and does not require the services of a Java Enterprise Edition (J2EE) application server. If desired, it can also be configured to run inside any Java-compliant servlet container. This increases the number of target environments that the OpenSource ESB can be deployed to.

System Failure Isolation

System failure isolation is an important principal achieved on the OpenSource ESB. This means there are no systems that the ESB interacts with that are required to be up for the ESB to be operational. For example all database configuration information (general configuration, system connection parameters, security information, etc.) is loaded into memory registries and accessed internally when needed for operation to allow the ESB to function properly even when its database goes down. While individual services will have critical path dependencies on external systems, the ESB runtime should continue to run as expected and provide use for services that do not interact with downed systems.

3.2 Logical Layers

The OpenSource ESB runtime is divided into the three logical layers: core, components, and client interaction points (CIPs). These logical layers interact with clients and service providers (or source systems) as detailed in the figure below:


Clients
Service Providers
Client Interaction Points
Components (Service Engines & Binding Components
Core (JBI Implementation & Framework

OpenSource ESB Logical Layers

Each layer in the architecture interacts only with other, adjacent layers. The layers are loosely coupled to minimize the impact on adjacent layers as changes occur. For example, ServiceMix is currently used as the JBI implementation (core module), but it could be exchanged for another implementation (e.g. OpenESB) without affecting the components that run on top of it.

Clients

Clients are systems that initiate interaction with the ESB using the SDA, EDA, or ETL pattern (in the case that a client is triggering an ETL transaction). Clients access services on the ESB through CIPs, normally communicating over open standard protocols such as SOAP or REST over HTTP.

Service Providers

Service providers are the source systems through which services located on the ESB are implemented. These service providers are called from respective binding components over various protocols, both open and proprietary. Interaction between clients and service providers never occurs directly, but always through NASA-ESB.

Client Interaction Points

Client interaction points provide the communication gateway for client interaction. Once a CIP receives a request from a client, it routes that request to the appropriate components (usually a service engine) for processing. Little or no custom code should be required for this layer as the bulk of the business logic for any service should be contained within a component. This layer also handles all client-based security (authentication, authorization, etc.).

Components

JBI Components are where most of the logic for services within the OpenSource ESB is contained. Service Engines (SE) provide the business logic, transformation, and coordination between other components. They can be either generic (XSLT, Rules Engine) or integration/service specific.

Binding components are used to make outgoing calls to service providers from the ESB. In addition to providing protocol service operation, binding components should provide built-in features such as archiving, event capturing, and statistics.

Core

The core layer encompasses the JBI implementation and a set of framework libraries that are globally accessible within the system. The JBI implementation provides the communication infrastructure for all component interaction (specifically the normalized message router) and supports all JMX management features defined within the JBI specification. The framework libraries are custom develop code that is reusable across many components and doesn't make sense to encapsulate as a component itself. An example of this type of code would be a system configuration library. Typically, component developers would not operate at this layer.

3.3 Implementation Details

3.3.1 CIP

The CIP layer as described above provides external access for calling clients. Each CIP type is implemented as a BC component but all communications flows from the outside in to the message router (in other words other components never call these CIP BCs). The following list contains the currently supported list of CIP transport types and the details of each implementation.

SOAP/REST over HTTP(s)

SOAP and REST over HTTP is supported with the HTTP BC. This component embeds the Tomcat [6] servlet container to support HTTP and runs the Apache Axis2 [7] web application to support SOAP and REST requests. The majority of client requests processed on the ESB occur over this transport. The Apache Axis2 stack was chosen to support SOAP and REST for the following reasons:

- **Speed –** Axis2 uses its own object model and StAX (Streaming API for XML) parsing to achieve significantly greater speed than earlier versions of Apache Axis.
- **Low Memory Foot Print –** Axis2 was designed ground-up keeping low memory foot print in mind.
- **Hot Deployment –** Capable of deploying Web services and handlers while the system is up and running.
- **Asynchronous Web Services –** Supports asynchronous Web services and asynchronous Web services invocation using non-blocking clients and transports.
- **Component-Oriented Deployment –** You can easily define reusable networks of Handlers to implement common patterns of processing for your applications, or to distribute to partners.
- **WS-*** – Provides support for the major WS-* specifications including but not limited to:
    - WS-Security, WS-Signature, WS-Encryption
    - WS-Notification, WS-BaseNotification, WS-Topics, WS-BrokerNotification
    - WS-Addressing, WS-Eventing, WS-Enumeration
    - WS-Policy*
    - WS-ReliableMessaging, WS-Reliability, WS-RM
- **WSDL Support –** Supports the Web Service Description Language, version 1.1 and 2.0.
- **Add-Ons –** Several Web services specifications have been incorporated including WSS4J [8] for security, Sandesha [9] for reliable messaging, and Kandula [10] which is an encapsulation of WS-Coordination, WS-AtomicTransaction and WS-BusinessActivity.

JMS

JMS synchronous messaging is supported using the ActiveMQ [11] JMS server. A JMS BC is used to retrieve JMS messages from a runtime defined message server and process them within the ESB. The

JMS BC run in either embedded or external model. In embedded mode the ActiveMQ server is started within the same JVM as the ESB runtime. In external mode, the BC connects to external ActiveMQ server(s) to retrieve incoming messages.

RAP (Reliable Asynchronous Processing)

JMS asynchronous messaging is supported using Reliable Asynchronous Processing. RAP is implemented in the ESB as a binding component. The basic workflow of the component is similar to the current persistence binding component with the main difference being ActiveMQ replaced with the RAP storage engine. The RAP storage engine is implemented as a library that runs in the same JVM as the RAP component. It uses a local hard drive as its primary storage mechanism. If local storage is unavailable it reverts to either SAN storage or database storage.

sFTP

An sFTP BC is used to support the SFTP protocol. The FTP server implementation used for this BC is from the MINA [12] project and is fully FIPS 140-2 compliant.

SCP (over SSH)

Secure copy over SSH is implemented as a BC using the SSHTools [13] project. This protocol implementation and FTP both allow the ESB to receive files internally that generally will trigger an asynchronous event to occur.

SAP RFC

The SAP Remote Function Call protocol is supported through the SAP server BC. This BC uses the JCo (Java Connector) library to support incoming calls from the SAP.

RMI

The ESB supports Java Remote Method Invocation (RMI) through the Agent component. In addition to supporting external client RMI calls, the Agent component is responsible for ESB to ESB communication in support of service location transparency.

3.3.2 Components

The OpenSource ESB runs a variety of components (both service engines and binding components). A distinction is made between service specific and common (reusable) components. Service specific components encapsulate the business logic of a service (transformation, routing, etc.) while common components implement a generic function reused by service components. The list below captures most of the common components developed for the OpenSource ESB platform. In addition to the common components developed locally, any JBI compliant component can be re-used on this platform.

Common Service Engines

XSLT SE

The XSLT service engine provides a generic interface for performing XSLT transformations. This component uses the Xalan [14] Apache library.

Scripting SE

The Scripting service engine provides a framework for running any JSR 223 [15] compliant scripting engine. This allows developers to create service logic using languages other then Java. Currently the Ruby, JavaScript, and Groovy programming languages are supported.

Rules Engine SE

The Rules Engine service engine allows for the declarative execution of rule sets based on trigger events from other components. The underlying Rules Engine is based on Ruleby [16].

Common Binding Component

SAP BC

The SAP BC allows for outbound calls to SAP Remote Function Calls (RFC) using the JCo (Java Connector) library.

Web Service BC

The Web service binding component allows other components to make generic web service calls using SOAP or REST over HTTP/S. The Apache Axis2 client library is used to make the Web service calls.

MDX BC

The MDX (Multidimensional Expression) binding component is a highly specialized component for making MDX queries to BW information cubes.

RAP BC

The Reliable Asynchronous Processing binding component is the primary mechanism supporting asynchronous messaging. It uses the RAP storage engine for all message stores and provides features such as throttling, holding, forwarding, message inspection and message manipulation.

Persistence BC

The Persistence BC allows for asynchronous processing of messages. It uses ActiveMQ for all message stores and provides features such as throttling, holding, and forwarding. All asynchronous integrations utilizing the Persistence BC will be transitioned over to the RAP BC.

Email BC

The Email BC allows for sending SMTP based messages.

LDAP BC

The LDAP BC allows for making LDAP queries using the Directory Service Markup Language [17] (DSML).

JDBC BC

The JDBC BC provides generic data access to any JDBC compliant database.

DCOM BC

The DCOM BC provides generic data access to any DCOM compliant service endpoint.

Additional BC Support

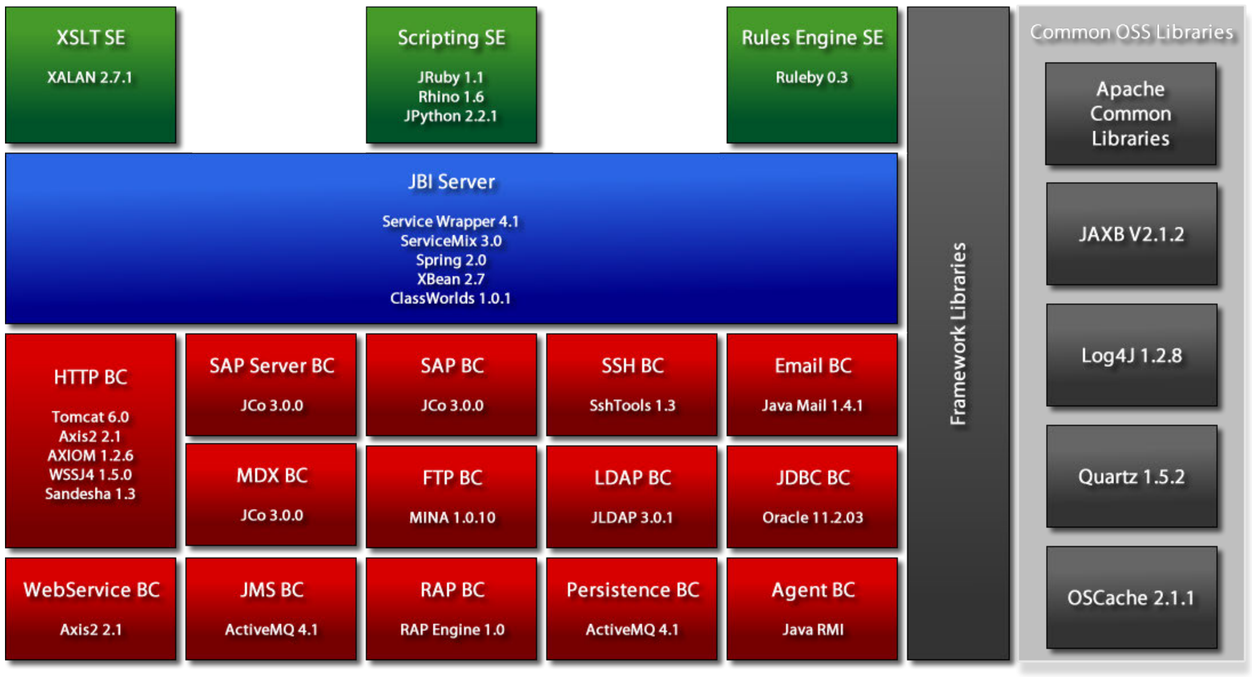
The OpenSource ESB provides the ability to utilize the standard reference ServiceMix binding components and/or develop custom binding components for any proprietary API system connector.

3.3.3 Core

The Core level includes the JBI implementation and the framework libraries. The JBI implementation is supported with the ServiceMix implementation. Any non-JBI features provided by ServiceMix have been avoided at the above layers to prevent implementation lock in. The framework libraries consist of common code re-used across the various layers and includes functionality for:

- Security
- Configuration
- Utilities
- JBI Support

The diagram below captures overall technical design of the OpenSource ESB. The majority of open source (and proprietary) projects and libraries that are currently used are listed at the various layers.



OpenSource ESB Technical Design

4. ETL Broker

NASA CC has developed an Extract Transform Load (ETL) broker runtime that runs on top of the Open ESB architecture. This runtime provides integration with systems that do not easily support the SDA and EDA integration patterns and require batch data replication from other systems (i.e. ETL). The broker runs off a simple table-driven interface which allows for the rapid implementation of ETL integrations with little to no development code. A common reusable set of ETL based connectors (for extracting and loading) have been developed and implemented as JBI Binding Components. These common components provide a solution to the majority of systems requiring ETL based integration. In cases where more complex integration logic is required, a custom ETL binding component can be developed to service this scenario.

4.1 Guiding Principles

The ETL broker platform for the NASA CC has been in use since 2001. The following guiding principles have been used to drive the design of this broker runtime.

OpenSource ESB Compliant

An important goal the ETL broker runtime is that it runs on top of the OpenSource ESB runtime. Extractors and loaders have been developed as JBI compliant binding components. The ETL flow controller has been developed as a JBI compliant service engine. The advantages of running ETL integrations on the same platform as our SDA and EDA based services are:

- Development Cohesion – Many times the same developers who develop components for SDA or EDA based services will also be tasked with developing ETL integration components. Having all of these pieces run on the same JBI architecture and interact in a similar fashion reduces the context switching and knowledge ramp up time between the two platforms.
- Release Management Cohesion – Updates to the ETL runtimes are executed in a similar fashion to the ESB runtime.
- Monitoring Cohesion – Operators who provide monitoring support of the production based runtimes use the same tools and procedures for diagnosing problems that occur.

Pattern Based Development

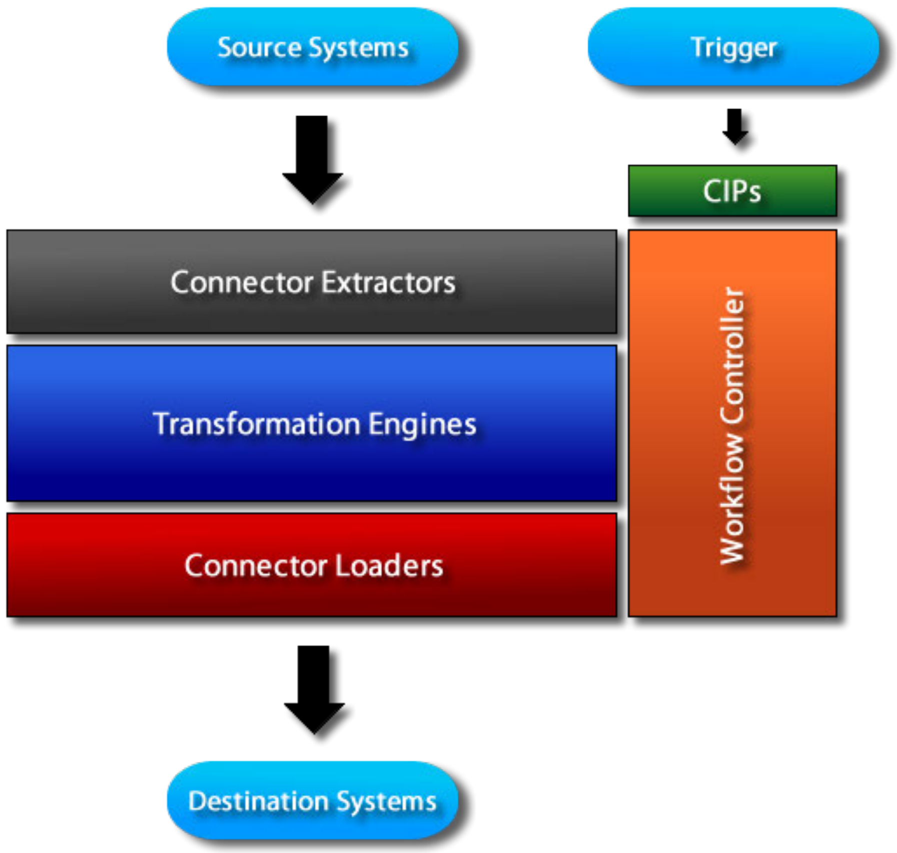
A very large majority of ETL based integrations fall into well established interaction patterns. A goal of the ETL platform is to take advantage of this regularity so developers can rapidly construct common ETL integration patterns and do not have to repeatable develop the same type of code. To support this goal a workflow engine has been developed where developers can wire together existing loaders and extractors and provide a minimal amount of transformation logic to complete development of common pattern based integrations.

Separation of Concerns

The ETL broker has been designed into four distinct layers (4.2 Logical Layers). This clear separation of concerns between the working pieces of the ETL broker allows new systems (source or destination) to quickly be supported on the platform as only the external communication with that system needs to be implemented (as an JBI BC loader or extractor).

4.2 Logical Layers

The ETL broker runtime is divided into the following logical layers: CIPs, workflow controller, connector extractors, transformation engines and connector loaders. These logical layers interact with source and destination systems as detailed in the figure below:



ETL Broker Logical Layers

Each layer in the architecture interacts only with other, adjacent layers. The layers are loosely coupled to minimize the impact on adjacent layers as changes occur.

Trigger

Triggers are events that initiate an ETL transaction. The trigger event contains no actual data to be processed but only contains the information required to start up the process. Triggers can be initiated from external systems that require ETL services or internally from the ETL broker itself (the CRON service engine is an example of an internal trigger).

CIPs (Client Interaction Points)

The CIP layer processes all trigger events and forwards the request on to the workflow controller to begin processing.

Workflow Controller

The workflow controller drives the interaction between the other ETL broker logical layers. It is configured using a workflow state diagram syntax and supports such constructs as serial and/or parallel execution, multi-condition based branching, and intelligent error resolution and notification.

The workflow controller is also responsible for handling the trigger events that initiate and ETL process. This layer is implemented as a JBI service engine.

Source Systems

Source systems are systems that contain the originating data. This data will be extracted from the source system and eventually put into one or more destination systems.

Connector Extractors

Connector extractors are responsible for extracting data from source systems. These extractors are implemented as JBI binding components.

Transformation Engines

Once the data has been housed internally by the connector extractors, the transformation engines are applied to the data to place it in an acceptable form for the destination systems. The transformation engines are implemented as JBI service engines.

Connector Loaders

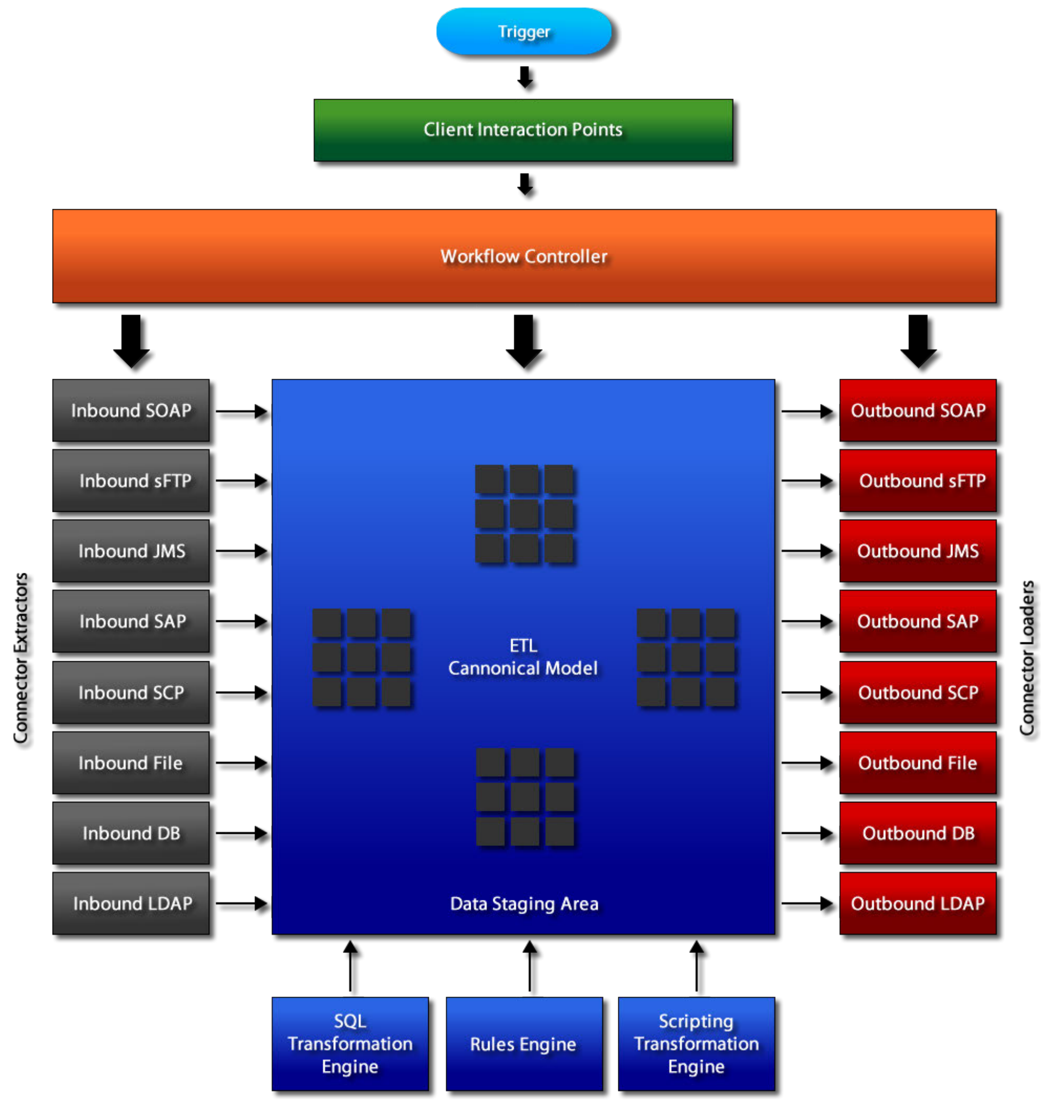
Connector loaders are responsible for loading the internally stored (and typically transformed) data into the destination systems. These loaders are implemented as JBI binding components.

Destination Systems

The target systems are systems that data (after processing) is placed into.

4.3 Implementation Details

This section will cover the implementation specifics of the ETL broker on top of the OpenSource ESB. The following diagram depicts the major pieces of this platform:



ETL Broker Technical Implementation

4.3.1 CIP

The CIP (Client Interaction Point) layer as described above provides external access for calling clients to initiate trigger that initiate an ETL transaction. The CIP layer is identical to the one described in 3.3.1 ESB CIP Layer.

4.3.2 Workflow Controller

The workflow controller is implemented as a JBI service engine. It communicates over the normalized message router to coordinate with the extractors, transformers, and loaders to complete an entire ETL transaction.

4.3.3 Connector Extractors

The connector extractors are responsible for extracting out data from source systems and placing the data into a common form in the internal data staging area. These extractors are implemented as JBI binding components. The following list contains the currently supported list of extractor transports:

SOAP/REST over HTTP(s)

The ETL SOAP extractor supports pulling data from any Web service (SOAP or REST). Once the data has been extracted, XSLT is used to transform the data into common ETL form.

sFTP

The sFTP extractor supports FTPS protocol for pulling files and loading them into common ETL form and is fully FIPS 140-2 compliant.

JMS

The JMS extractor can be configured to work with any JMS compliant client library. ActiveMQ is currently in use.

SAP

The SAP Remote Function Call (RFC) protocol is used with inbound SAP extractor. It supports loading data from SAP into the common data staging area.

SCP (over SSH)

Secure copy over SSH is implemented as a BC using the SSHTools project. This protocol implementation and FTP both allow the ETL broker to receive files internally into the data staging area for processing.

File

The file extractor supports pulling files directory off of mapped drives on the ESB.

DB

Data from a relational database can extract using the DB extractor. Any relational database were a JDBC compliant driver is available can be used.

LDAP

LDAP can be used as a source (only) system that the LDAP extractor can pull data from.

Additional Support

The OpenSource ESB provides the ability to utilize the standard reference ServiceMix connector extractors and/or develop custom connector extractors for any proprietary API system connector.

4.3.4 Data Staging Area

The data staging area is an internal data source that the ETL broker uses to house data from source systems that needs to be (eventually) loaded into destinations systems. This data store is currently implemented as an Oracle database. A common ETL schema is used to store all information. By using this central data store (and the common schema), connector extractors and connector loaders can be decoupled from each other and paired up as needed. It also provides an easily accessible location to perform transformation on by the various transformation engines.

4.3.5 Transformation Engines

Once the data from source systems is housed internally in the data staging area, transformations can be executed against this data to prepare it for targeted destination systems. The following transformation engines are supported:

SQL Transformation

The SQL transformation engine provides internal database transformations using PL/SQL. These transformations have very high performance and are ideal for transforming very large datasets.

Scripting Transformation

The scripting transformation engine allows developers to express transformations using a full fledged programming language (currently Java or Ruby). Although the scripting transformation engine does not achieve the same performance as the SQL transformation, in many scenarios the added flexibility and expressiveness is worth this trade off.

4.3.6 Rules Engines

Once the data from source systems is housed internally in the data staging area, various rules engines can be applied against these datasets (either pre or post transformation) to execute a set of tasks. The Ruleby service engine is currently used to support this feature.

4.3.7 Connector Loaders

The connector loaders are responsible for taking the data from the data staging area and loading them into destination systems. These loaders are implemented as JBI binding components. The following list contains the currently supported list of loader transports:

SOAP/REST over HTTP(s)

The ETL SOAP loader supports sending data to any Web service (SOAP or REST).

sFTP

The sFTP loader supports the FTPS protocol for placing data files to destination systems and is fully FIPS 140-2 compliant.

JMS

The JMS loader can be configured to work with any JMS compliant client library and is used to send JMS messages that contained the extracted (and transformed) data.

SAP

The SAP Remote Function Call protocol is used with outbound SAP loader to send data to SAP.

SCP (over SSH)

Secure copy over SSH is implemented as a BC using the SSHTools project. This protocol implementation and FTP both allow the ETL broker to take internally stored data and place them as external files

File

The file extractor supports placing files into directories off of mapped drives on the ESB.

DB

Data can be loaded into an external relational database using the DB loader. Any relational database were a JDBC compliant driver is available can be used.

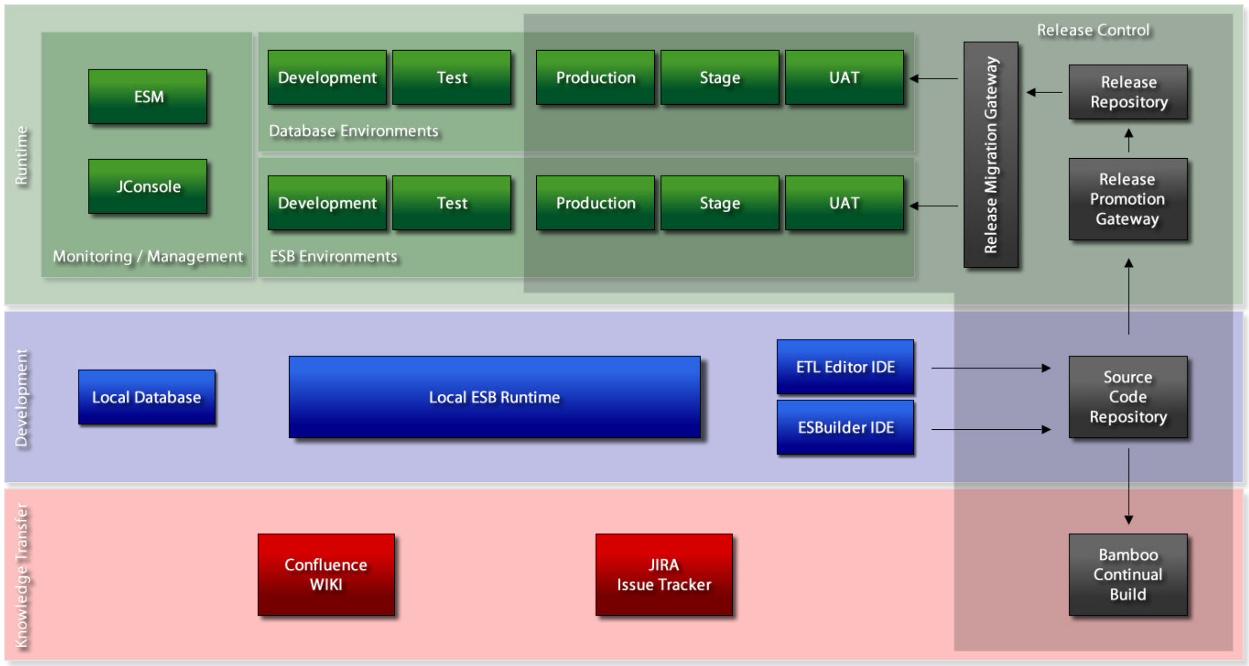
LDAP

LDAP can be used as a source (only) system that the LDAP extractor can pull data from.

Additional Support

The OpenSource ESB provides the ability to utilize the standard reference ServiceMix connector loaders and/or develop custom connector loaders for any proprietary API system connector.

5. OpenSource ESB Landscape



OpenSource ESB Landscape

The diagram above captures all the tools and systems used to develop services and systems on the OpenSource ESB platform. The arrows on diagram capture the flow of code throughout the landscape. The landscape contains the following levels: knowledge transfer, development, runtime and release control.

5.1 Landscape Levels

5.1.1 Knowledge Transfer Level

The knowledge transfer level is where all tools related to knowledge transfer reside. The Atlassian [18] suite of products is currently used to support this level.

Confluence

Confluence is used as the OpenSource ESB wiki. The wiki is used to capture all tutorials, lessons learned, best practices, and coding standards for the OpenSource ESB platform.

Jira

Jira is used to track and manage all software changes and configuration across the NEACC landscape.

Bamboo

Bamboo is the continual build server used to verify all unit tests for components and framework level code. Code coverage reports are generated with Clover.

5.1.2 Development Level

All local development of components and framework level code is at this level. Developers run a local instance of the OpenSource ESB that they use to test out new development until its ready to be promoted the runtime level.

Local ESB Runtime

This local runtime mimics the functionality of the ESB instances in the runtime level. For the majority of the developers on a Window's OS, the ESB is installed as local service that can be started and stopped as needed.

Local Database

The local OpenSource ESB uses a local database installed on the developers machine. NASA CC developers are currently using Oracle Express and SQL Developer.

ESBuilder IDE

The ESBuilder is a set of plugins for Eclipse [19] developed specifically for working with the OpenSource ESB platform. These plugins allow developers to quickly develop components in addition to installing their components onto the local ESB instance.

ETL Editor IDE

The ETL Editor is a set of plugins for Eclipse developed specifically for working with the ETL broker platform. These plugins allow developers to quickly develop ETL integration flows in addition to installing their components onto the local ESB instance.

Source Code Repository

The source code repository is used to store and provide history for all OpenSource ESB source code. Subversion is currently used to provide this.

5.1.3 Runtime Level

The runtime level is where clients and external systems interact with the OpenSource ESB.

ESB Environments

The ESB environments contain the OpenSource ESB instances. The following environments are supported:

Development – The development environment is managed by the developers and they add and update components as needed. This is the environment that initial development with clients is conducted in and is the least stable environment.

Test – The test environment is managed by the leads on the development team. Components are added and updated when needed but the constraints on these updates are tighter and provide for a more stable development environment for external clients and systems.

UAT – The UAT environment provides an iterative release environment with unique support for both minor (daily, weekly, monthly) and major releases (semi-annual i.e., 15.1, 15.2). The UAT environment is managed by a separate operations group and provides a controlled environment for iterative end-to-end integration testing.

Stage – The stage environment provides an identical environment (both software and hardware configuration) to the production environment. This environment is managed by a separate operations groups and provides the runtime for all acceptance testing before being promoted to production.

Production – The production environment provides a highly scalable and fault tolerant platform supporting the OpenSource ESB and ETL broker instances. This environment is managed by a separate operations group that provides 24x7 support.

Database Environments

Database environments mirror the development, test, UAT, stage and production ESB environments. Oracle is the database product currently used for development activities.

Monitoring/Management

Monitoring and management is used by developers and operators to provide support for the ESB environments. The following tools are used:

ESM – The ESM (Enterprise Service Monitor) is a web application developed specifically to support the OpenSource ESB platform. This tool provides real-time transaction visibility, error and alter notification, trending reports, and system status.

JConsole – JConsole interfaces with the OpenSource ESB using JMX [20]. The OpenSource ESB provides a fully configurable JMX interface for making on the fly adjustments.

Release Repository

Since code deployment to the staging and production environments occurs outside of the development group, a separate release repository is used that contains the runtime binaries need to update the ESB with. This repository is populated with compiled code (that has passed all unit tests) from the source code repository.

Release Promotion Gateway

The release promotion gateway is the tool used by development leads to promote code (components, framework, etc.) from the source code repository to the release repository. After the code is checked out from the source code repository, it must compile appropriately (if valid) and pass all unit tests before the binaries are published to the release repository.

Release Migration Gateway

The release migration gateway is the tool used by operators to publish binaries to the OpenSource ESB instances. This gateway pulls all binaries from the release repository.

References

[1] ServiceMix http://servicemix.apache.org

[2] UDDI http://www.cio.com/article/2433285/service-oriented-architecture/why-uddi-sucks html

[3] An ESB should have services that can be easily found and consumable

[4] Amdahl's Law http://en.wikipedia.org/wiki/Amdahl's_law

[5] SEDA Staged Event Driven Architecture http://www.eecs.harvard.edu/~mdw/papers/seda-sosp01.pdf

[6] Tomcat Web Application Server http://tomcat.apache.org

[7] Apache Axis2 http://ws.apache.org/axis2

[8] WSS4J (WS-Security) http://ws.apache.org/wss4j

[9] Sandesha (WS-RM) http://ws.apache.org/sandesha

[10] Kandula (WS-Coordination, WS-AtomicTransaction, and WS-BusinessActivity) http://ws.apache.org/kandula

[11] ActiveMQ JMS Server http://activemq.apache.org

[12] MINA (FTP Project) http://mina.apache.org

[13] SSHTools http://sourceforge net/projects/sshtools

[14] Xalan (XSLT Parser) http://xml.apache.org/xalan-j

[15] Scripting for the Java Platform JSR 223 http://jcp.org/en/jsr/detail?id=223

[16] Ruleby (Ruby based rules engine) http://ruleby.org/wiki/Ruleby

[17] DSML (Directory Services Markup Language) http://www.oasis-open.org/committees/dsml/docs/DSMLv2.doc

[18] Atlassian http://www.atlassian.com

[19] Eclipse http://www.eclipse.org

[20] JMX (Java Management Extensions) http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement